Network Working Group
RFC #684
NIC #32252
April 15,1975

A Commentary on Procedure Calling as a Network Protocol

Richard Schantz

BBN-TENEX

Preface

This RFC is being issued as a first step in an attempt to stimulate a dialog on some issues in designing a distributed computing system. In particular, it considers the approach taken in a design set forth in RFC #674, commonly known as the "Procedure Call Protocol" (PCP). In the present document, the concentration is on what we believe to be the shortcomings of such a design approach.

Note at the outset that this is not the first time we are providing a critical commentary on PCP. During the earlier PCP design stages, we met with the PCP designers for a brief period, and suggested several changes, many of which became part of PCP Version 2. We hasten to add, however, that the nature of those suggestions stem from an entirely different point of view than those presented here. Our original suggestions, and also some subsequent ones, were mainly addressing details of implementation. In this note the concern is more with the concepts underlying the PCP design than with the PCP implementation.

This note is being distributed because we feel that it raises certain issues which have not been adequately addressed yet. The PCP designers are to be congratulated for providing a detailed written description of their ideas, thereby creating a natural starting point for a discussion of distributed system design concepts. It is the intent of this note to stimulate an interaction among individuals involved with distributed computing, which could perhaps result in systems whose designs don't preclude their use in projects other than the one for which they were originally conceived.

The ideas expressed in this RFC have benefited from numerous discussions with Bob Thomas, BBN-TENEX, who shares the point of view taken.

Introduction

While the Procedure Call Protocol (PCP) and its use within the National Software Works (NSW) context attacks many of the problems associated with integrating independent computing systems to handle a distributed computation, it is our feeling that its design contains flaws which should prevent its widespread use, and in our view, limit its overall utility. We are not voicing our objection to the use of PCP, in its current definition, as the base level implementation vehicle for the NSW project. It is already too late for any such objection, and PCP may, in fact, be very effective for the NSW implementation, since they are proceeding in parallel and have probably influenced each other. Rather, we are voicing an objection to the "PCP philosophy", in the hope of preventing this type of protocol from becoming the de-facto network standard for distributed computation, and in the hope of influencing the future direction of this and similar efforts.

Some of the objectionable aspects of PCP, it can be argued, are differences of individual preference, and philosophers have often indicated that you cannot argue about tastes. We have tried to avoid such arguments in this document. Rather, we consider PCP in light of our experience in developing distributed systems. Considered in this way, we feel that PCP and its underlying philosophy have flaws which make it inappropriate as a general purpose protocol and virtual programming system for the construction of distributed software systems. It is our opinion that PCP is probably complete in the sense that one can probably do anything that is required using its primitives. A key issue then, is not whether this function or that function can be supported. Rather, to us an important question is how easy it is to do the things which experience has indicated are important to distributed computing. In addition, a programming discipline dedicated to network applications should pay particular attention to coercing its users away from actions which systems programming in general and network programming in particular have shown to be pitfalls in system implementation.

A Point of View

At the outset, we fully support the aspects of the PCP design effort that have gone into systematizing the interaction and agreements between distributed elements to support inter-machine computing. This includes the definition of the various types of replies, the standardization of the data structure format for inter-machine exchange, and the process creation primitives which extend the machine boundaries. Such notions are basic and must be part of any distributed system definition. Our main concern is not with these efforts.

Rather, we take exception to PCP's underlying premise: that the procedure calling discipline is the starting point for building multi-computer systems. This premise leads to a model which has a central point for the entire algorithm control, rather than a more natural (in network situations) distributed control accomplished by cooperating independent entities interacting through common communication paths. While the procedure call may be an appropriate basis for certain applications, we believe that it can neither directly nor accurately model the interactions and control structures that occur in many distributed multi-computer systems.

Much of what follows may seem to be a pedagogic argument, and PCP supporters may take the position of "who cares what you call it, its doing the same thing". Our reply is that it is very important to achieve a clear and concise model of distributed computation, and while the PCP model does not require "poor implementation" of distributed systems, neither does it make "good implementation" any easier, nor does it prohibit ill-advised programming practices. A model stressing the dynamic interconnection of somewhat independent computing entities, we feel, adheres more to the notions of defensive programming, which we have found to be fundamental to building usable multi-machine implementations.

The rest of this RFC discusses what we feel to be some of the shortcomings of a procedure call protocol.

Limitations of Procedure Calling Across Machines

First and foremost, it is our contention that procedure calling should not be the basis for multi-machine interactions. We feel that a request and reply protocol along with suitably manipulated communication paths between processes forms a model better suited to the situation in which the network places us. In a network environment one has autonomous computing entities which have agreed on their cooperation, rather than a master process forcing execution of a certain body of code to fulfill its computing needs. In such a configuration, actions required of a process are best accommodated indirectly (by request) rather than directly (by procedure call), in order to maintain the integrity of the constituent processes.

Procedure calling is most often a very primitive operation whose implementation often requires only a single machine instruction. In addition, it is usually true that procedure calling is usually not within the domain of the operating system. [The Multics intersegment procedure calling mechanism may present an exception to this, until linkage is complete. In the remote PCP case, however, linkage can never be complete in the sense of supporting a fast transfer of control between modules]. Processes and communication paths between processes, however, are undeniably operating system constructs. In an environment where local procedure calling was "cheap", it would be ill-advised to blur the

distinction between a local (inexpensive in time and effort) and a remote procedure call, which obviously requires a great deal of effort by the "PCP system", if not by the PCP user. It also seems to be the case that the cost of blurring the local/remote distinction at the procedure call level will be found in the more frequent use of a less efficient local procedure calling mechanism. Interprocess communication, on the other hand, (at least with regard to stream or message oriented channels and not just interrupt signals) is generally regarded as having a significant cost associated with it. Message sending is always an interprocess action, and requires system intervention always. There is not as substantial a difference between the IPC of local processes and the IPC of remote processes, as between local and remote procedure calling. PCP is suggestive of a model in which processes exist that span machine boundaries to provide inter-machine subroutine calling. Yet the PCP documentation has not advocated the notion of a process that spans machine boundaries, and rightfully so since such a creation would cause innumerable problems. Since procedure calling is more suitable as an intra-process notion, it seems to be a better idea to take the interprocess communication framework and extend it to have a uniform interpretation locally and remotely, rather than to extend the procedure calling model. It is also our contention that a model which relies on procedure calling for its basis does not take into account the special nature of the network environment, and that such an environment can be more suitably handled in a message passing model. Furthermore, we feel that programming as a whole, even purely local computing, will benefit from paying more attention to such areas as reliability and robustness, which have been brought to the forefront through experience with an oftentimes unreliable network and collection of hosts. An IPC model, by emphasizing the connections between disjoint processes, seems to reinforce the idea that distributed computing is accomplished by joining separate entities, and that defensive programming and error handling techniques are appropriate. Since PCP is, we think, for distributed system builders, and not for the end user (e.g. an RSEXEC user), avoiding the network, interconnection issues, and relative costs, may be counter-productive if the goal is to achieve usable network systems.

In a similar vein, the entire notion of inter-machine procedure calling underlies a model which in effect has extended the address space of a single process. That is, there is a single locus of algorithm control (although perhaps not a single locus of execution). While this model may well serve the needs of a "local" computation where the parts are strongly bound together, our experience in building working distributed systems has shown the utility of a model which has multiple loci of control and execution. In such a model, it is through agreements on the method and type of information interchange and synchronization, that a computation is carried out, rather than at the singular direction of a central entity. In a model that has distributed control and execution, we feel a process will be in a better position to naturally cope with the many vagaries that necessarily arise in a network environment.

The unmistakable trend in systems programming is toward inviolable (protected) process structures with external synchronization as a means of coping with complex debugging tasks and the difficulty of making system changes. This trend is better supported, we feel, by a message passing rather than a procedural model of computation. Furthermore, we feel that network programming techniques should be applied to local computation, not the other way around.

Some Particulars

In the following list, we try to be more specific with respect to particular situations where we think the PCP concept may be weak as the basis for a network programming system. For some of these examples to be meaningful, the reader should be fairly familiar with the PCP documents issued as RFC 674.

1. Recovery from component malfunction may be very difficult to handle by a process that is not the central control (i.e. a process which is being manipulated by having its procedures executed). Is the situation where there is network trouble, for example, to be modeled by a forced procedure call to some error recovery routine? It is precisely such situations where distributed control serves as a better model. Consider the act of introducing an inferior to another acquaintance and then supplying the new handle as a parameter of a subsequent procedure call in the inferior. The inferior's blind use of the parameter to interact with the other process illustrates the manipulative aspects of a superior. The inferior never really is aware of a new communication path to a new process. The inferior environment (as maintained by the PCP "system") has been changed by the superior, with no active notification of the inferior. Certainly this makes user coded error recovery somewhat awkward.

2. Such process manipulation may at times violate the principles of modular programming. In this vein, it seems beneficial to be able to debug separately the pieces of a computation and then worry only about their synchronization to achieve a totally debugged system. With PCP in its fullest sense, the danger of error propagation seems greater because of the power of a process to cause execution of an arbitrary procedure and to read/write remote data stores without the active participation of the remote process.

3. Can we assume a proper initialization sequence if our procedures are called remotely? Must every procedure contain the code to check for the propriety and correct sequencing of the call? A model in which each remote process is an active computing element seems better able to

conveniently apply protective standards to the code and data it encompasses.

4. PCP doesn't model long term parallel activity in a convenient fashion, as is required to handle various asynchronous producer/consumer process relationships. The synchronization is geared more to a one-to-one call and return, rather than to the asynchronous nature and multiple returns for a single request, as exhibited by many network services. In addition, low priority, preemptable background tasks are hard (impossible?) to model in a procedure call environment.

5. Communication paths are not treated as abstract objects which are independent from the actual entities they connect, and hence they cannot be utilized in some useful ways (e.g. to carry non PCP messages). Also with respect to treating communication paths as objects, there is no concept of passing a communication path to an inferior (or an acquaintance), without having to create a new "connection" (whether or not this turns out to be a physical channel). The ability to pass communication paths is often useful in subcontracting requests to inferior processes. To do this within PCP requires the cooperation of the calling process (i.e. to use the new connection handle), which again seems to violate the concepts of modular programming. The alternative approach in PCP is to have the superior relay the subsequent communications to its created inferior, but the effort involved would probably prohibit the use of this technique for subcontracting.

6. PCP seems too complicated to be used for the type of processing which requires periodic but short (i.e. a few words exchanged) interactions. An example of such interactions is the way the TIP uses the TENEX accounting servers (see RFC #672). Furthermore, PCP is probably much too complex for implementation on a small host. In that regard, there does not seem to be a definition of what might constitute a minimum implementation for a host/process which did/could not handle all of what has been developed.

7. In the PCP model, it may become awkward or resource consuming for a service program to do such things as queue operations for execution at a later time (persistence) or at a more opportune time (priority servicing mechanism). Such implementations may require dummy returns and modification of the controlling fork concept, or maintenance of processing forks over long periods of inactivity.

8. It is not always true that a process connecting (splicing) to a service should be able to influence the service process environment in any direct way. How can a service process in PCP prevent a malicious user fom splicing

to it and then introducing it to an arbitrary number of processes, thereby overflowing the table space in that process. All of that could have been done without ever executing a single instruction of user written code. This difficulty is a consequence of the PCP notion of having one process manipulate the environment of another without its active participation in such actions.

9. Doesn't the fact that the network PCP process implementation is so much neater than the TENEX PCP process implementation (since TENEX doesn't have a general IPC facility) suggest that message passing and communication facilities supported by the "system" provides a sound basis for multi-process implementations, and that perhaps such facilities should be primitively available to the distributed system builders who will use PCP?

10. There is a question of whether PCP is an implementation virtual machine (language), or an application virtual machine (language). That is, is PCP intended to be used to implement systems which manage distributed resources, or as an end product which makes the network resources themselves easier to use for the every day, ordinary programmer (e.g. makes the network itself transparent to users). One gets the feeling that the designers had both goals, and that neither one is completely satisfied. If the former goal is taken, we believe that most of the complexities (e.g. network trouble, broken connections, etc.) and possibilities (e.g. redundant implementation, broadcast request, etc.) of network implementations are not provided for adequately. In this view, the NSW framework (Works manager, FE) is the distributed system that utilizes the PCP implementation language. We do not see how the use of PCP in this context provides for either an extra-reliable system through component redundancy, or a persistent system which can tolerate temporary malfunctions. If one subscribes to this view, then it doesn't seem right that the objects that run under the created system (i.e. the tools that run under the PCP implemented Front End, Works Manager, and TBH monitor) should also be aware of or use PCP. If one considers the latter goal, that PCP implements a virtual machine to be presented to all programmers for making distributed resources easy to use, then it is clear that PCP with its manifest concern for object location does not provide for the desireable properties of network transparency.

Our conclusion is that procedure calling is not the appropriate basis for distributed multi-computer systems because it can neither directly nor accurately model the network environment. The PCP virtual programming system may be inadequate for implementing many distributed systems because the complexities and possibilities unique to the network environment are not provided for at this basic level.